RIOT -- A Simple Graphical Chip Assembly Tool

Stephen Trimberger
California Institute of Technology
Pasadena, California 91125

James Rowson
VLSI Technology, Inc.
Santa Clara, California

ABSTRACT

Riot is a simple interactive graphical tool designed to facilitate the assembly of cells into integrated systems. Riot supplies the user with primitive operations of connection -- abutment, routing and stretching -- in an interactive graphic environment. The designer retains full control of the design, including the assignment of positions to instances of cells and the choice of connection mechanism. The computer takes care of the tedious and exacting implementation detail, guaranteeing that connections are made correctly. The powerful connection primitives give the user of Riot the ability to quickly assemble a custom chip from a collection of low-level cells. This document provides a discussion of the motivation for Riot and a description of the Riot chip assembly system, its capabilities and its use.

INTRODUCTION

Custom integrated circuit layout can be split into two major parts: *cell design*, the development of the low level cells making up the "leaves" of the hierarchical tree; and *composition*, assembly of those cells into larger cells and systems [Rowson 1980]. Errors in composition are harder to find than errors in cell design, since they belong to no specific part of the design, but rather to the assembly as a whole. Composition errors are more costly than cell design errors also, since they often go unnoticed until late in the design cycle.

Current graphics systems address low-level cell design [Calma 1979], and rely on the simple graphical primitives used in cell design to perform composition. Graphical systems allow blocks to be oriented and positioned, but composition requires operations to position one instance adjacent to another and to logically connect connectors on instances. These features are usually unavailable in an interactive graphic system, so users must verify connections with extensive checking.

Most powerful composition tools currently being investigated are language based. The composition of instances is specified by textual invocation and connection. Although the language form is very powerful, the textual specification is tedious and error prone in general systems, since composition continues to be primarily a graphical positioning task. Highly automated language-based systems allow designs to be expressed simply in an abstract manner, but place severe restrictions on the floorplan of the resulting chip.

Riot is a simple interactive graphical composition tool. Riot provides primitives necessary for composition, so that designs can be made quickly and easily. Riot's primitives are powerful enough to be useful for composition, yet simple enough that control of the layout is still in the hands of the designer. The user of Riot can creates instances of cells and connect cells by abutment, simple routing, or stretching.

Environment

Riot runs on the Caltech graphic workstation which consists of a "Charles" color terminal, a high resolution color raster display device; a CRT terminal; a Xerox *mouse* pointing device; and a Hewlett-Packard 7221A four-color pen plotter; all driven by a DEC LSI-11. Riot also runs on the low-cost GIGI workstation, which consists of a DEC GIGI color terminal and a Summagraphics BitPad™ pointing device (see figure 1).

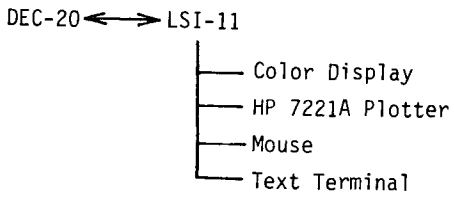

Figure 1a. "Charles" Terminal Color Graphic Workstation

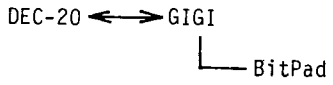

Figure 1b. GIGI Terminal Color Graphic Workstation

Riot is written in SIMULA and runs on the Caltech Computer Science Department DECsystem-20. The SIMULA environment includes a powerful symbolic high-level language debugger and memory management facilities with dynamic memory allocation and a garbage collector. Riot consists of approximately nine thousand lines of code, including the shared low-level objects package (500 lines) and graphics package (4000 lines).

The interface between Riot and all other design tools at Caltech is through three standard data formats: Caltech Intermediate Form (CIF) [Sproull 1980], the Sticks Standard (Sticks) [Trimberger 1980], and Composition Format. CIF, a geometrical format, is generated by many design tools, including Bristle Blocks [Johannsen 1979], LAP [Lang 1979]

BitPad is a trademark of the Summagraphics Corporation, Fairfield, Connecticut.

and PLA generators. A user extension was added to CIF to indicate connector locations so that Riot's logical connection operations could be performed on CIF cells. CIF is also used for mask generation. Sticks, a symbolic layout format, is generated by the graphical and textual symbolic layout systems, and is also used as input to simulation. The composition format is used by Riot to save an editing session. It contains a description of composition cells including the hierarchy description, locations of instances, locations of connectors on the composition cells, and references to files which contain the leaf cells used in those compositions.

Riot Definitions

Riot deals with a modification of the restricted hierarchy called the *separated hierarchy.* [Rowson 1980]. The hierarchy is built of *cells*, which are collections of integrated circuit data. There are two kinds of cells in Riot: *leaf cells* on the leaves of the hierarchical tree, consisting of primitive geometry or Sticks and instances of cells; and *composition cells* in the interior of the tree, which consist only of instances of other cells. Riot allows the user to assemble composition cells and leaf cells into larger composition cells.

Composition cells are described internally by a bounding box, a list of connectors, and a list of instances of cells contained in the cell. A *connector* consists of a location on or inside the bounding box of the cell, and the layer and width of the wire that makes that connection. An *instance* represents the contents of a cell placed at a given location with a specified orientation and array replication count.

Riot makes a *connection* by placing instances so that the connectors on the instances touch. Riot handles connection in the positional sense, not in the logical sense: a connection is the result of appropriate positioning. Therefore, once a connection is made, it can be easily (perhaps accidentally) destroyed. Riot guarantees that connections will be made correctly, but does not guarantee that those connections will be maintained. This simplified handling of connection has limited the usefulness of Riot, as discussed later in this paper.

Riot's connection operations connect connectors on one instance to connectors on many instances, moving the single instance to touch the others or stretching the single instance, depending on the kind of connection being made. This one-to-many restriction simplified the routing algorithm immensely and eliminated the need for heuristics in a many-to-many abutment. A many-to-many connection can still be made by defining a cell which contains one of the sets of cells, and connecting that one to the other many.

Riot Overview

In some ways, Riot is very similar to existing simple graphical design systems. Internally, Riot has a list of cells that the user may edit. These cells are simpler in Riot than in typical graphics systems, because they contain only instances of cells. In Riot, the user may create an instance of a cell, which adds that instance to the list of instances in the cell under edit. The user may move, orient and replicate instances, which make the appropriate changes to the instance transform or the replication counts.

The connection commands in Riot makes it different from existing graphics systems. The connection operations require that Riot keep a list of pending connections. The list is shown on the screen constantly, and the user may add to and delete from this list. The connection commands examine the connection list and may change the positions of the instances or create an instance of a routing cell in order to make the connections.

Another difference between Riot and existing graphic systems is Riot's inability to make primitive geometry. This limitation is not fundamental to the operation of the tool, however the separation of leaf cell design and composition tasks has made Riot much simpler both from a systems designer's viewpoint and a user's viewpoint. The limitations imposed by this separation are not significant. We have many tools to create the cells which Riot manipulates.

Riot has two command interfaces: one textual, one graphical. The textual command interface, accessed with the keyboard, is used primarily to modify the editing environment. Textual commands store and retreive cells on disk, set plotting parameters, generate hardcopy plots of cells, set defaults for routing operations, and invoke the graphical command editor to modify a composition cell.

The user edits a cell with the graphical command interface by pointing at items on the graphic display. The Riot display screen is divided into three pieces (figure 2): a large editing area next to two small menu areas along the right edge of the screen. The editing area shows the contents of the cell under edit. The upper menu area contains the names of the cells which are currently defined and which may be instantiated. The lower menu contains graphical editing commands which are invoked by pointing at them. These graphical commands include the commands to move, orient, and connect instances as well as commands to modify the display characteristics.

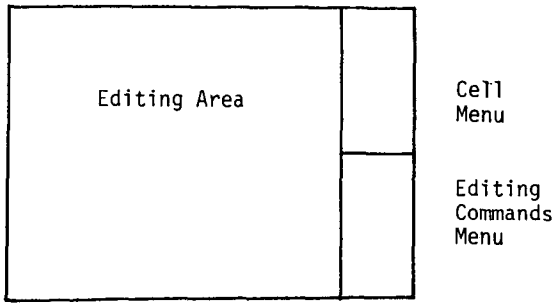

Figure 2. Riot Display Organization

RIOT OPERATIONS

Riot has commands for four different tasks: interface to the environment, creation of instances, connection of instances, and completion of a cell. This section discusses the four kinds of commands.

Interface to the Environment

Riot's interface commands access external cell definitions, generate hardcopy plots of cells in Riot, and modify the display and routing characteristics. Riot can read leaf cells defined in CIF or Sticks, and composition cells defined in composition format. Riot writes composition format files which are converted to CIF for mask generation or to Sticks for simulation. Since Riot is an interactive graphical tool, commands exist for zooming and panning the display. Commands also exist to delete and rename cells, and to produce a hardcopy plot of a cell.

Creation of Instances

A user of Riot creates an instance by first selecting the cell in the cell menu by pointing at its name in the cell menu, then giving the CREATE command in the editing window. The user can optionally specify replication counts in x and y directions to create arrays, replication spacing for arrays, rotation by multiples of 90 degrees, and mirroring of the instance. An array instance is treated like any other instance, and shows the gridding due to the replication of the cell in the array and all the connectors on the outside edge of the array. Although the array spacing can be modified by the user, array elements must connect properly by abutment, because Riot allows no access to interior connectors on arrays.

Internally, Riot keeps an instance as a pointer to the defining cell with a transformation, replication counts, and replication spacings. An instance is represented on the screen by the bounding box and connectors of the defining cell positioned, oriented, and replicated by the instance information (figure 3). The size and color of the connector crosses indicates width and layer of the wire making the connection inside the cell. Optionally, instances can be displayed with their cell names and connector names to facilitate identification.

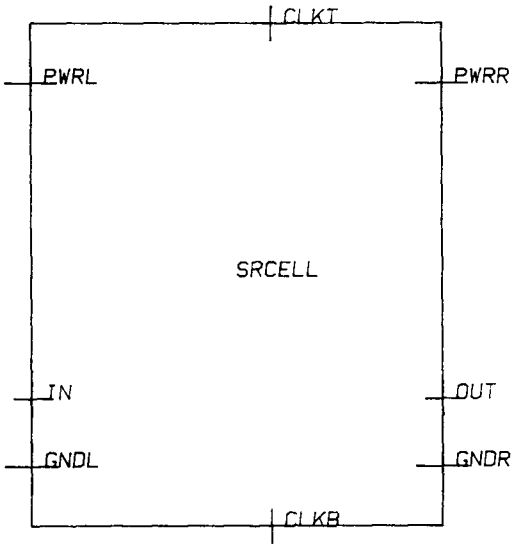

Figure 3. Riot's View of a Cell Instance with Bounding Box and Connectors

Connection of Instances

Riot was built to facilitate making connections between instances, and this is where it differs from typical interactive graphic systems. Riot has simple positioning and orientation commands similar to those found in traditional graphics systems. These commands are used in Riot to prepare instances for connection but are not used to make the actual connection of instances. Riot's connection commands connect instances by abutment, routing or stretching, guaranteeing connection and allowing the designer to select the connection method which best fits the design at that point.

Connection of instances is done in two parts: first, the connections to be made are specified, then a *connection specification command* is given to actually make the connection. A *connection* is a link from a connector on one instance to a connector on another instance. Riot checks that the connectors to be joined are on the same layer and that they are opposed. That is, that they connect top to bottom or left to right.

Internally, Riot keeps pointers to the instance and to the connector on both sides of the connection until the connection specification command is given. When that command is given, the instance pointers are used to identify the instances to be connected, and Riot uses the instance and connector information together to generate two vectors of points to be used for routing or for generating stretch sizes. After the connection specification command, the logical connection information is thrown out.

Riot provides three ways to express connections: the user may specify merely that the instances are to be abutted, which is used if a cell has no connectors; the user may identify specific connections to be made; or the user may specify a bus-type connection in which all connections are made from one instance to another. The connection information includes the instance from which connections are to be made (the *from* instance) and the instance(s) to which connections are to be made (the *to* instance). In normal operation, possible changes such as movement or stretching, will be made to the *from* instance.

There are three connection specification commands: abut, route and stretch. All three connection types are provided in Riot so the user can select the kind of connection that is most appropriate at each place in the design.

Abutment is used primarily if there are no connectors to guide the connection. Abutment makes the bottom or left edge match, depending on the relative positions of the instances before the ABUT command (figure 4). If specific connections to connectors exist, Riot will attempt to match the specified connections during the abutment. If the connections cannot be made by the abutment, a warning message is produced. An option of the abutment command allows instances to be overlapped to share a common pair of connectors. This feature is frequently used to share power or ground lines in adjacent instances.

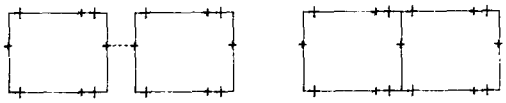

Figure 4. Connection by Abutment

Riot is able to make simple multi-layer *river-routed* connections. A multi-layer river-route is a routed connection between parallel sets of points where no routes change layers and no two routes on the same layer cross. The Riot river router cannot turn corners, and it ignores objects in the path of the route. This route is much simpler than routing found in industrial systems. The limitations are not too severe because the route described here is only one piece of the total connection of connectors on the entire chip. The global connection is done incrementally and interactively in Riot, and consists of routed connections, stretched connections and abutted connections.

When the ROUTE command is given, the connectors on the *from* and *to* instances are used to specify starting and ending locations of the route. The wire widths of the route are determined by the widths of the connectors on the instances being connected. The routing algorithm attempts to route all wires to the desired locations in a single routing channel. If some wires are blocked, another channel is added and the route is continued in the new channel. This repeats until the connection is completed.

Riot then makes a new Sticks cell containing the river route wires and places an instance of that route cell next to the *to* instance. The *from* instance is moved to abut the other side of the river route instance, thereby using the least amount of space possible for the route (figure 5). Optionally, the user may specify that a route be made without moving the *from* instance. This feature is used to make connections between two instances which are already positioned and should not move.

The routing cells made in Riot are treated just like other cells. They are entered in the list of cells in the cell menu, and may be instantiated, moved, and deleted by the user.

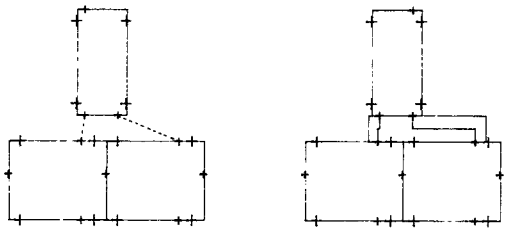

Figure 5. Connection by Routing

In many situations, a stretched connection uses less space than a routed connection. In a *stretched* connection, the locations of the connectors on the *to* instance are used to determine the needed separations of the connectors on the *from* instance to make the connection by abutment. If the *from* instance is defined in Sticks form, the new constraints on the connector positions are put into the Stick file, making a new cell. The new cell is passed through the Stick optimizer in REST [Mosteller 1981], which moves the connectors to the constrained locations. Riot then removes the old instance and inserts an instance of the new cell into the cell under edit (figure 6). The new locations of the connectors allow the instances to be abutted without routing.

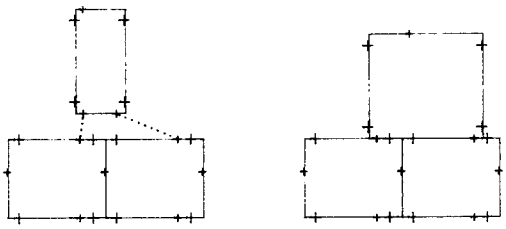

Figure 6. Connection by Stretching

Finishing a Cell

A composition cell created by Riot includes those connectors from its instances which lie on its bounding box. The route command can be used to "bring out" connectors from the inside of the cell to the edge of the composition cell. When an attempt is made to route the connectors on an instance past the bounding box of the cell, a simple straight-line route cell is made for those connectors to the edge of the cell, and an instance of that cell is placed to make the connection.

RIOT EXAMPLE

This section takes a step-by-step walk through a Riot editing session, showing some of the features of the tool. The chip being assembled in this example is a four-bit sequential logical filter: a function defined on a series of inputs, x as

$$f_n = \sum_{i=1}^{4} c_i x_{n-i}$$

where the c_i constants are supplied from off-chip and all sums and products are Boolean. A rough initial floorplan is shown in figure 7, showing how the designer wishes to lay out the design. This floorplan determines which cells are needed, how they must connect to one another, and gives an initial guess at critical paths in the design.

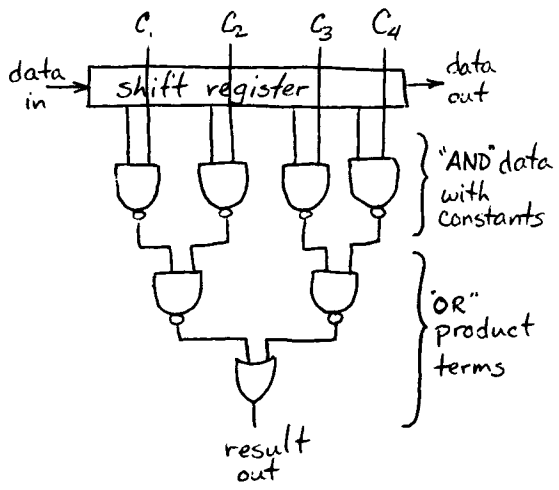

Figure 7. Rough Floorplan for Logical Filter

The cells to be assembled in Riot are shown in figure 8. The input and output pads were taken from a library of CIF cells. The shift register cell, NAND and OR gates were laid out in REST, and are defined as symbolic layout in Sticks. Therefore, the pads cannot be stretched by Riot and all connections to them will have to be made by routing, but connections to the other cells can be made by stretching if the designer wishes to do so.

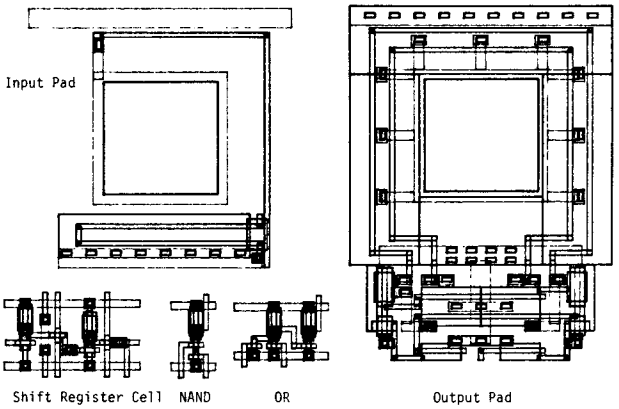

Figure 8. Leaf Cells for Logical Filter

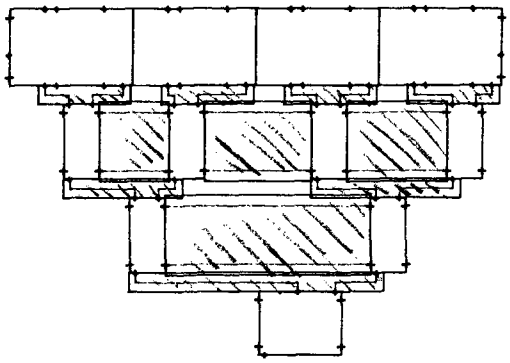

Figure 9a. Logical Filter Logic Connected with Routing.

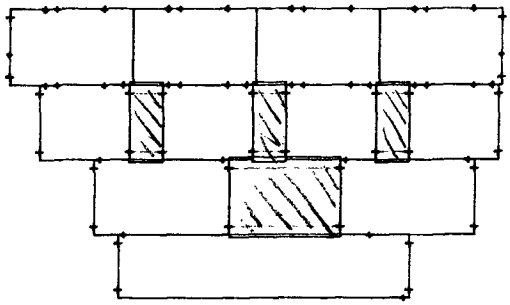

Figure 9b. Logical Filter Logic Connected with Stretching.

The first step is to generate the shift register array. The array elements abut, making the shift register chain connections as well as power and ground connections. Next, two stages of NAND gates provide the ANDing of the constant terms and the first level of ORs, then routing is done to the OR gate. Connections to these gates are routed in figure 9a. Alternatively, the designer may save area by stretching the gates, eliminating the routing area (figure 9b).

In figure 9, the shaded areas are routing areas. The relative dearth of routing area in the stretched version is misleading, since some of the area is wasted inside the cells. The important space savings is in the vertical direction since no routing channels are needed to connect the NAND and OR gates.

The definition of the logic portion is finished by routing connections to the edge of the cell so they show as connectors on the larger cell. Pre-defined *pipe fittings* aid complex routes for power, ground and clock lines. Pad routing is done in pieces with Riot's routing command, and the completed chip geometry is shown in figure 10.

This logical filter chip is, of course, of insignificant complexity, and is used only to show how Riot's capabilities can be used to assemble a chip. The same techniques that allow this small custom chip to be assembled quickly can be used to significantly shorten the assembly time for larger custom chips.

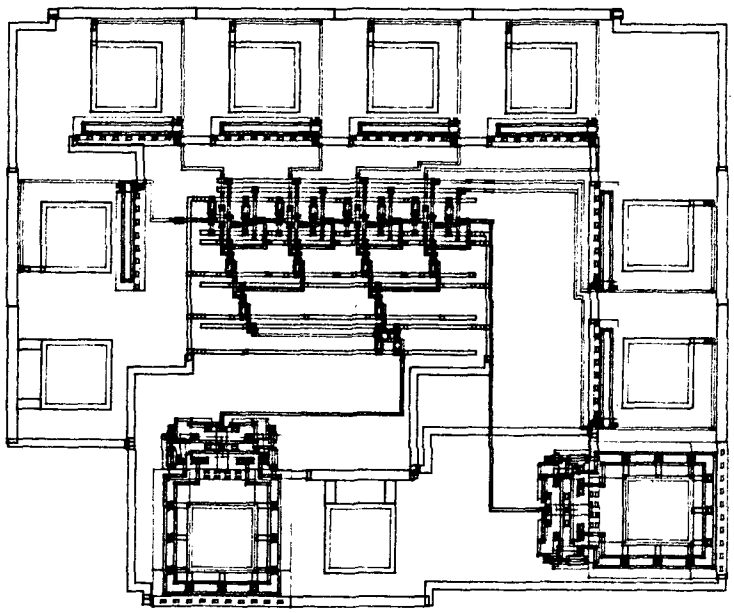

Figure 10. Logical Filter Chip Geometry

OBSERVATIONS, LIMITATIONS AND CONCLUSIONS

Riot was developed very quickly with shortcuts taken to simplify the software design. Riot represents about four man-months of effort. The result is a simple tool, augmenting the designer's capabilities, allowing designs to be created very quickly, and leaving difficult problems to the designer. This simplification is not without a price. This section discusses the main problems with Riot and the steps which are being taken to remedy those problems in Riot and in future systems.

Connection by Positioning

A fundamental problem with Riot is its treatment of connection as geometrical position, which leaves the burden of maintaining logical connection to the designer. Riot determines positions of instances and routes by the positions of connectors on instances, but once the instances are positioned to make the connection, the fact that the two pieces are connected is lost, and the user is free to move the pieces in whatever manner is desired. When the logical connection is lost, some of the designer's intent is lost, also.

The existence of a connection is not remembered, so the connections can easily be inadvertantly destroyed by further editing. In this interactive system, though, connections are not often accidentally destroyed, and when they are destroyed, the error is fairly easy to detect and correct However, the mere possibility of missed connections requires checking by users and has severely limited the usefulness of Riot.

Modification of Leaf Cells

When an existing leaf cell is modified, the locations of connectors are often changed also, and since Riot's connections are positional happenstance and Riot allows instances to overlap, connections will no longer be made properly and no warning message will be generated. Thus, the user is forced to re-edit major portions of the chip by hand to re-connect instances.

Riot includes an inexpensive solution to this problem, the REPLAY. Riot saves the commands given by the user and can re-run an editing session if some of the input files have changed. The replay file uses instance names and connector names to identify connections, and the positions are re-calculated, thereby avoiding the problems with differently-shaped cells. The replay also enables users to recover an abnormally-terminated editing session or an accidentally-deleted file. The replay mitigates the problem of logical connection being destroyed during editing, but does not solve it. The replay is not an acceptable long-term solution to this important problem -- connections must be preserved.

Conclusions

Riot has been used in the preparation of several small project chips, and appears to be a very good tool for such small designs. Riot reads both CIF and Sticks, simplifying the task of combining parts of designs from different systems. Riot provides a general interface to symbolic layout, allowing symbolic cells to be easily modified to fit into different environments on the chip, and on other chips. This greatly improves the usefulness of our symbolic layout systems.

The advantages and disadvantages of Riot have spurred further tool development in this area. Without further investigation, we can say that a tool of this type must maintain logical connections, a feature which is lacking in Riot. New graphical chip assembly tools are under construction at Caltech which will remedy this problem.

Even without maintaining connection, Riot has shown itself to be a powerful design aid. Riot allows designers to assemble test projects quickly with a minimum of interference from the tool. The simple connection commands are adequate to design large chips when the designer can use them as he wishes in different situations. Since the designer has the option at every point to choose the kind of connection operation, the design can be optimized at the level of assembly, not just at the level of layout. Riot is a valuable addition to the integrated system designer's arsenal.

REFERENCES

Calma (1980). "GDS II Product Specification", Calma Interactive Graphics Systems, Sunnyvale, CA.

Johannsen, D. (1979). "Bristle Blocks: A Silicon Compiler", Proceedings of the 16th Design Automation Conference, 1979.

Lang, D. (1979). "LAP: A SIMULA Package for IC Layout", Computer Science Department Technical Report, California Institute of Technology, 1979.

Mosteller, R. (1981) "REST -- A Leaf Cell Design System", Master's Thesis, California Institute of Technology, 1981.

Rowson, J.A. (1980). "Understanding Hierarchical Design", PhD Thesis, California Institute of Technology, 1980.

Sproull, R. and Lyon, R. (1980). "The Caltech Intermediate Form for LSI Layout Description" in C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison Wesley, New York, 1980.

Trimberger, S. (1980). "The Proposed Sticks Standard", Computer Science Department Technical Report, California Institute of Technology, 1980.